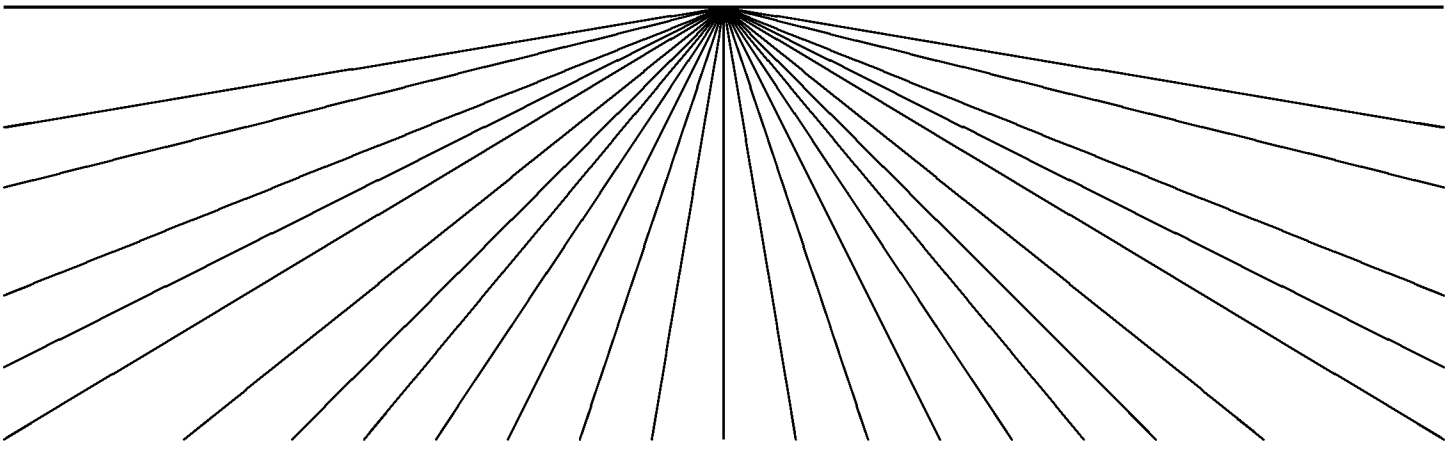
facultad de informática

universidad politécnica de madrid

An Automatic Translation Scheme from CLP to AKL

Francisco Bueno
Manuel Hermenegildo

TR Number CLIP7/95



An Automatic Translation Scheme from CLP to AKL

Technical Report Number: CLIP7/95

June 1995

Keywords

Constraints, Concurrent Constraint Programming, AKL, Program Transformation

Acknowledgements

This work was funded in part by ESPRIT project 7195 "ACCLAIM" and by CICYT projects TIC93-0975-CE and TIC93-0737-C02-01 IPL-D.

Abstract

The Andorra Kernel language scheme was aimed, in principle, at simultaneously supporting the programming styles of Prolog and committed choice languages. Within the constraint programming paradigm, this family of languages could also in principle support the concurrent constraint paradigm. This happens for the Agents Kernel Language (AKL). On the other hand, AKL requires a somewhat detailed specification of control by the user. This could be avoided by programming in CLP to run on AKL. However, CLP programs cannot be executed directly on AKL. This is due to a number of factors, from more or less trivial syntactic differences to more involved issues such as the treatment of cut and making the exploitation of certain types of parallelism possible. This paper provides a translation scheme which is a basis of an automatic compiler of CLP programs into AKL, which can bridge those differences. In addition to supporting CLP, our style of translation achieves independent and–parallel execution where possible, which is relevant since this type of parallel execution preserves, through the translation, the user–perceived "complexity" of the original program.

Contents

1 Introduction

Previous work [BH92] showed that it is possible to perform program transformations to bridge the existing gap between the sequential and concurrent paradigms of logic programming. In [BH92] a transformation from Prolog to AKL was proposed, which had the additional advantage of allowing the latter to fully exploit the Independent And-Parallelism (IAP) present in Prolog programs. Exploitation of this kind of parallelism has the advantage of preserving the computational complexity of the original sequential programs [HR95], thus allowing a rational exploitation of the concurrent underlying machinery.

When extending the transformation techniques to the constraint logic programming paradigm [JL87, Col90, Hen89] — CLP, some issues have to be initially solved. First, the notion of independence in CLP had to be clarified. In this line of work, the notion of Constraint Independence has been proposed [dlBG94] (see also [dlBHM94]), which extends the IAP notions of traditional logic programming and prove equivalent to them when restricting the constraint system to that of (pure) logic programming, i.e. Herbrand. Second, compile-time tools based on the above notions had to be developed in order to capture the independence of goals, allowing such transformation. In [dlBBH95] a complete automatic parallelizing compiler for CLP is presented, based on constraint independence notions and suitable analysis technology for CLP.

We present in this report an extension of the work reported in [BH92] which makes use of the program analysis and transformation methodology of [dlBBH95]. The resulting transformation achieves similar results for CLP to those of [BH92] for Prolog. Although AKL does not share the same semantics as the sequential logic programming paradigm, its functionality is available in AKL. Thus, we can define a program transformation from CLP into a subset of AKL which shares similar semantic properties. Therefore available analysis technology for CLP is still applicable.

Because of that reason, our aim is not to take advantage of AKL properties to provide the best possible translation, but rather to bridge the gap between CLP and AKL. Our main objective is to show how the independence principle can be moved to paradigms other than the sequential one by using program transformation, and with similar methods than in the sequential case. The transformed program will thus respect the semantics and complexity of the original program, perhaps reducing the search space complexity because of exploiting independence. This is an important result also, and additionally, because of the possibility of achieving *stability* earlier (and at compile–time) in the framework of AKL — an important control principle for efficiency, by detecting goal independence in the source language.

2 The Agents Kernel Language

The Agents Kernel Language — AKL, is a logic programming language which subsumes important parts of both the sequential and the concurrent logic programming paradigms, offering possibilities for both committed–choice style indeterminism (and thus concurrency) and also search–oriented nondeterminism. Its computational model is based on the extended Andorra model — EAM [War90, HJ90], and thus allows a wide range of capabilities for exploiting parallelism, being this a main objective in the conception of the EAM. AKL is therefore quite a powerful language; nonetheless it can be argued that it has some drawbacks, stemming from its powerful execution model: it doesn't have as simple a semantics as SLD–resolution, and it does put quite a burden on the programmer in requiring certain specification of control. This motivates considering a translation to it of programs with a simple semantics and minimal specification of control, such as CLP programs. Such a translation is also useful for simply running CLP programs in AKL which may have been written for a CLP system. The language and computational model of AKL have been fully described in [Jan94]. In the following we will review the AKL computational model restricted to a subset of that in [Jan94], with the purpose of, based on an understanding of this, extracting the correct rules for a translation of CLP which achieves the desired results.

Procedures are defined in AKL by the guarded clauses of a definition. Thus, clauses are divided into two parts: the *guard* and the *body*, separated by a *guard operator*. Guard operators are: *wait* (`?`), *conditional* (`->`), and *commit* (`|`). An additional conditional guard operator has also been implemented in AKL (`!`), which we will discuss later.

Definition 1 (AKL program) *Let $\vec{x}$ be a tuple of variables. The following grammar gives the syntax of the (subset of) AKL that we consider:*
$Program ::= Procedure.Program \mid \epsilon$
$Procedure ::= Clause.Procedure \mid Clause$
$Clause ::= Head \mid Head\text{:-}\ Body \mid Head\text{:-}\ Body\ GuardOp\ Body$
$Head ::= Atom$
$Body ::= Literal \mid Literal, Body$
$Literal ::= Atom$
$GuardOp ::=$ `|` $\mid$ `?` $\mid$ `->` $\mid$ `!`
$Atom ::= \mathtt{p}(\vec{x})$ ■

Note the following syntactical restrictions on the above grammar:

- Heads for the clauses of the same procedure definition relate to the same predicate.
- Each clause is expected to have one and only one guard operator.
- All clauses in a procedure definition have to have the same guard operator. Thus,

if any of the clauses is not guarded, the guard operator of its companions is assumed and positioned just after the clause neck.

- A wait operator is assumed, and in the above mentioned position, where no other operator can be assumed using the above mentioned rules.

Guards in AKL can be *deep*. Guards are regarded as part of clause selection. This means that a clause body is not entered unless head unification succeeds and its guard is completely solved. Then, execution proceeds by expansion of the present *configuration* by application of a rule of the computation model. Configurations describe computation states.

Definition 2 (configuration) *Let c denote a constraint. The following grammar defines AKL configurations:*
$Configuration ::= AndBox \mid OrBox$
$OrBox ::= \texttt{or}(AndBox^*)$
$AndBox ::= \texttt{and}_c(Goal^*)$
$Goal ::= atom \mid ChoiceBox$
$ChoiceBox ::= \texttt{choice}(GuardedGoal^*)$
$GuardedGoal ::= Configuration\ GuardOp\ Goal^+$
$GuardOp ::= \texttt{|} \mid \texttt{?} \mid \texttt{->} \mid \texttt{!}$ ■

A choice–box will always have a unique associated guard operator. Choice–boxes are fundamental in AKL as they represent unfolding of procedures. Atomic goals will be rewritten into the choice–boxes corresponding to their procedure definitions. Guards will be executed in and–boxes and solved until completion, the and–boxes then being empty. And–boxes have a constraint part: their *environment*. Environments will hold the solution of empty and–boxes. When an atomic goal is a constraint, the *constraint* rule adds it to the environment of the configuration.

$$\texttt{and}_c(R, G, T) \Rightarrow \texttt{and}_{c \cup G}(R, T)$$

For other program atoms the *reduction* rule allows unfolding of procedure definitions. If the atomic goal G is defined by n guarded clauses of the form $A_i \% B_i$, for some guard $\%$,

$$\texttt{and}_c(R, G, T) \Rightarrow \texttt{and}_c(R, \texttt{choice}(\texttt{and}_c(A_1)\%B_1, \ldots, \texttt{and}_c(A_n)\%B_n), T)$$

Environments also allow capturing local computations of guards. Guards are executed in independent environments, and thus, isolated from the parent configuration. Therefore, solutions of guards are not "visible" to the rest of the computation until they are *promoted*. Solutions can be either consistent or inconsistent with the parent environment. Configurations will then be either successfully promoted or failed. A single solved remaining alternative in a choice–box is promoted by the *promotion* rule,

if its solution is consistent with the parent environment. If the guard operator is either conditional or commit, quietness of the solution is required.

$$\texttt{and}_c(R, \texttt{choice}(\texttt{and}_d()\%B), T) \Rightarrow \texttt{and}_{c \cup d}(R, B, T)$$

If the solution of a solved guard is inconsistent with the environment ($c \cup d \vdash false$), the guard is failed. Failed guards in a choice–box are eliminated by the *guard failure* rule.

$$\texttt{choice}(S_1, \texttt{and}_{false}()\%B, S_2) \Rightarrow \texttt{choice}(S_1, S_2)$$

A choice–box can then become empty by failure of all alternatives. Failure is then propagated by the *goal failure* rule.

$$\texttt{and}_c(R, \texttt{choice}(), T) \Rightarrow \texttt{and}_{false}()$$

If a guard is solved in a choice–box which shows other alternatives, a nondeterminate promotion of the guarded–goal with solved guard has to be done. The *choice splitting* rule does this. A control restriction is added in this case: the configuration has to be *stable*. We will comment on this later.

$$\texttt{and}_c(R, \texttt{choice}(S_1, \texttt{and}_d()\%B, S_2), T) \Rightarrow$$

$$\texttt{or}(\texttt{and}_c(R, \texttt{choice}(\texttt{and}_d()\%B), T), \texttt{and}_c(R, \texttt{choice}(S_1, S_2), T))$$

Finally, if when solving a guard, it shows alternatives, these will end by applying nondeterminate promotion in an or–box. Or–boxes in guards are distributed over the surrounding choice–box by the *guard distribution* rule.

$$\texttt{choice}(R, \texttt{or}(S_1, S_2)\%B, T) \Rightarrow \texttt{choice}(R, S_1\%B, \texttt{or}(S_2)\%B, T)$$

Note that in this case, a singleton or–box is equivalent to its contained and–box: $\texttt{or}(\texttt{and}_c(Goal)) \leftrightarrow \texttt{and}_c(Goal)$.

Pruning is achieved by the *condition* and *commit* rules. The condition rule may be applied if the "right" part of the alternatives (T) is not empty; the commit rule if there is at least one alternative (either R or T is not empty). Quietness of the solved guard is also required. The *noisy cut* rule is the same as the condition rule except for the quietness requirement.

$$\texttt{choice}(R, \texttt{and}_c()\texttt{->}B, T) \Rightarrow \texttt{choice}(R, \texttt{and}_c()\texttt{->}B)$$

$$\texttt{choice}(R, \texttt{and}_c()\,\texttt{|}\,B, T) \Rightarrow \texttt{choice}(\texttt{and}_c()\,\texttt{|}\,B)$$

Pruning rules in AKL (except for the noisy cut) are applicable only to *quiet* solutions. A solution for the guard of a conditional or commit guarded–goal (clause) is quiet if it does not further restrict (or constrain) variables outside its own configuration.

The rules in the AKL computational model allow rewriting of configurations leading to valid configurations from valid ones.

Definition 3 (AKL computation [Jan94]) *Given an AKL program P and an initial goal g, a computation for g in P is a finite or infinite transition sequence of the form* $\mathtt{or}(\mathtt{and}_{true}(g)) \Rightarrow \ldots$ *which, if finite, ends with a configuration of the form* $\mathtt{or}(\mathtt{and}_{\delta_1}(), \ldots, \mathtt{and}_{\delta_n}())$, *where the δ_i are the solutions to g, and the computation is successful; if the or–box is empty, the computation is failed.* ∎

An AKL computation is deterministic, except for the application of the choice splitting rule, or nondeterminate promotion. The application of this rule in fact amounts to copying[1] the "continuation" configuration. Because of this, if it were the case that in this configuration a rule other than the nondeterminate promotion rule could be applied, then it should have to be applied separately in all the copies, after promotion. Therefore, the situation will be inefficient. This same situation can occur because of external constraints, which could prune the search space of the configurations involved, making determinate promotion applicable where it was not before. Hence, the stability restriction.

A configuration is stable if no (deterministic) rule is applicable within it, and no possible changes in its environment will lead to a situation in which a (deterministic) rule can become applicable. If otherwise, copying the continuation will force to apply the applicable rules in both copies. Note that this restriction causes (deterministic) promotion to be preferred over nondeterminate promotion. In this manner, configurations (goals) becoming deterministic upon execution are executed first, as by the Andorra principle.

Stability is central to AKL efficiency. However, the global stability condition is difficult to check, and in fact, it is undecidable. Instead, a local sufficient condition is usually employed: if a configuration, where no other rule than nondeterminate promotion is applicable, can not produce constraints over variables with scope outside the configuration itself, then no sibling configuration can "affect" it. Indeed, configurations can not affect each other, and in this sense, are *independent*. Note that the condition effectively amounts to the independence notions which have been studied for constraint programming in [dlBG94, dlBHM94]. Therefore, a transformation based on independence, as the one we will propose, would deal effectively with stability, too.

3 Translating Definite Clauses

We will first show how AKL gives the basic CLP functionality, i.e. SLD–resolution. We consider definite clause syntax as given by Definition 4. In the AKL transformed programs we will make guards and guard operators explicit, for the sake of clarity. Definite clause programs can be considered AKL programs, with no other transformation than bridging syntactical differences and making unification constraints explicit.

[1] Although we refer to "copying," part of the continuation in the configuration could in principle be shared [War90].

Definition 4 (CLP program) *Let c be a constraint, and $\vec{t}$ a tuple of terms. The following grammar defines the syntax of CLP programs:*
$Program ::= Clause.Program \mid \epsilon$
$Clause ::= Atom \mid Atom\text{:-}\ B$
$B ::= C \mid Body \mid C.Body$
$C ::= c \mid c, C$
$Body ::= Literal \mid Literal, Body$
$Literal ::= Atom$
$Atom ::= \mathtt{p}(\vec{t})$ ∎

For convenience, we have defined CLP programs in such a way that constraints can be separated from the rest of the clause bodies. Except where explicitly said, when referring to a body in a CLP program, we will mean the literal part of it. The classical left–to–right operational semantics of Prolog and CLP [JL87] will be considered. We will denote the computation states in this semantics by a goal and a store, as in $\langle g, \theta \rangle$, where g is the goal and θ the store.

Algorithm 1 (definite clause AKL program) *Given a definite clause program P, its corresponding AKL program is given by $dc2akl(P)$. Let $\vec{t}$ denote a tuple of terms and $\vec{x}$ a tuple of variables. Let true be a constraint always satisfied. Then $dc2akl(C.P) = dc2aklc(C).dc2akl(P)$, where:*

$$dc2aklc(C) = \begin{cases} p(\vec{x})\text{:-}\ true?\vec{x} = \vec{t}, B & \text{if } C \text{ is } p(\vec{t})\text{:-}\ B \\ p(\vec{x})\text{:-}\ true?\vec{x} = \vec{t} & \text{if } C \text{ is } p(\vec{t}) \end{cases}$$

Additionally, for every predicate `p/n` *defined in P a clause of the form $p(\vec{x})$:- $true?fail$ is added to $dc2akl(P)$.* ∎

The reason for adding dummy clauses to every definition is to eliminate predicates which, from its definition, will be deterministic [Jan94]. Every AKL procedure definition will then have at least two clauses. This will forbid application of the promotion rule until choice splitting is done, and will also guarantee stability of configurations.

Given a definite goal $g_1, \ldots, g_n$, it will then be executed as an AKL initial configuration $\mathtt{or}(\mathtt{and}(g_1, \ldots, g_n))$, where the reduction rule will be applied yielding:

$$\mathtt{or}(\mathtt{and}_c(\mathtt{choice}(G_1, \ldots, G_{m_1}), \ldots, \mathtt{choice}(G_n, \ldots, G_{m_n})))$$

where the corresponding procedure definition for each g_i has m_i clauses of the form $G_j = true?B_j$, with $j = i, \ldots, m_i$. Note that guards are all solved.

Because $\forall i, m_i \geq 2$ no determinate computational rule can be applied. As it also happens that guards do not add constraints to the environment, the resulting and–box is stable. A choice split of any of the choice–boxes will then be performed. However, we will be able to identify the computational step which applies this rule to the leftmost choice–boxes with a corresponding SLD–derivation.

Theorem 1 (AKL subsumes SLD–resolution) *Given a definite clause program P and its corresponding AKL program $dc2akl(P)$, then for every SLD–derivation of P there exists an AKL computation of $dc2akl(P)$ which has the same solution.*

Proof: (Outline) Consider an AKL computation in which choice split is always performed on the leftmost choice–box of the configuration. Since for the program $dc2akl(P)$ no determinate computational rule is applicable, and all and–boxes are stable, applying choice split to the leftmost choice–box results in an AKL computation which resembles SLD–resolution. Thus, this computation yields the same solutions. ∎

4 Translating CLP Constructions

In addition to SLD–resolution, CLP programs have the cut (!) pruning operator, side–effects, and meta–logical predicates. For the latter, no corresponding builtins are given in AKL. For the cut and side–effects, we will show how to extend the above transformation.

Even this straightforward step is nontrivial, as we shall soon see. This is due mainly to the semantics of cut in both CLP and AKL, cut being a guard operator in the latter. With the restrictions required for guard operators to achieve both syntactic and semantic correctness in AKL, we find problems in the following constructions:

- syntactical restrictions:
  - definitions of predicates in which a pruning clause appears,
  - clauses in which more than one cut appears;
- semantic restrictions:
  - if-then-elses, where the cut has a local pruning effect,
  - pruning clauses where the cut is regarded as *noisy* (i.e. attempts to further restrict variables outside its scope),
  - side–effects and meta–logical predicates, which should be sequentialised.

For the translation we consider CLP syntax as given above and regard the cut as an atom. Thus, the first thing to be noticed is the translation of if-then-elses, which are not considered in the syntax given. If-then-elses can be folded into new procedure definitions, where the cut can be given a global scope within the definition, and thus it can be directly considered as an AKL guard operator.

4.1 Translation of Cut

It is obvious that the cut can be translated either to the conditional guard operator, or the (non quiet) cut operator of AKL. We will show that the first option shows

advantages over the second one. But first, let us consider the syntactical restrictions on guard operators. One of them is that only one guard operator is to be allowed in a clause. Therefore repeated cuts in the same body (which are otherwise strongly discouraged as a matter of style and declarativeness) have to be folded out using the technique sketched below.

Example 1 *For the CLP program to the left, the AKL program to the right has single guard operators in all clauses.*

```
p(X,Y):-  test(X), !,                p(X,Y):- test(X) -> p1(X,Y).
          test(Y), !,
          accept(X,Y).               p1(X,Y):- test(Y) -> accept(X,Y).
```

Assume that for a given program P, we have that `pN/n` is a new predicate name, not appearing in P, taken from an infinite set of names, all of them distinct. Given a CLP clause C, let $cut(C)$ be true if there is an atom `!` in $body(C)$.

Algorithm 2 (unravelled cuts) *Let P be a CLP program, and let $\vec{x}$ denote a tuple of variables, and $\vec{t}$ a tuple of terms. We obtain a CLP program $P' = fold(P)$, where:*
$fold(C.P) = unravel(C).fold(P)$
$fold(\epsilon) = \epsilon$
and

$$unravel(C) = \begin{cases} \mathtt{p}(\vec{t})\text{:- } left(B), \mathtt{pN}(\vec{x}).unravel(\mathtt{pN}(\vec{x})\text{:- } right(B)) & \text{if } cut(C) \text{ and} \\ \quad C = \mathtt{p}(\vec{t})\text{:- } B(\vec{x}) \text{ and } left(B) \neq \epsilon \text{ and } right(B) \neq \epsilon & \\ C \quad \text{otherwise} & \end{cases}$$

and

$$left(B) = \begin{cases} \epsilon & \text{if } B = !, B' \\ A, left(B') & \text{if } B = A, B' \text{ and } A \neq ! \\ B & \text{otherwise} \end{cases}$$

and

$$right(B) = \begin{cases} B' & \text{if } B = !, B' \\ right(B') & \text{if } B = A, B' \text{ and } A \neq ! \\ \epsilon & \text{otherwise} \end{cases}$$

. ∎

A second syntactic restriction is that all AKL clauses in a procedure definition are forced to have the same guard operator. We can achieve this by simple foldings of the CLP definitions.

Example 2 *For the CLP program to the left, the AKL program to the right shows the same guard operator in all definitions.*

```
p(X,Y):- q(X), r(Y).                 p(X,Y):- q(X), r(Y).
p(X,Y):- test(X), !, output(Y).      p(X,Y):- p1(X,Y).
```

```
p(X,Y):- s(X,Y).
p(X,Y):- t(X,Y).
```

```
p1(X,Y):- test(X) -> output(Y).
p1(X,Y):- p2(X,Y).

p2(X,Y):- s(X,Y).
p2(X,Y):- t(X,Y).
```

Note that clauses before the pruning one will have an (assumed) wait operator and clauses after that one (and that one itself) will have an (assumed) conditional operator. Guard operators can be made explicit in the same way as in the previous section by introducing *true* in guards. Note that, had the program not been rewritten, the rules for assuming guard operators would have put a conditional operator in the first clause, which is obviously not the correct translation. Note also that successive foldings of the procedure definition have to be done as clauses with or without cut appear.

We formalise this transformation also on the CLP side. Assume `pN/n` given as before, let $rename(\texttt{pN}, P)$ be a program P' where all clause heads are renamed to `pN`.

Algorithm 3 (folded cuts) *Given a CLP program P, let it be composed by different procedure definitions D such that $P = D.P'$, and P' is a CLP program made up of procedure definitions, too. Let $\vec{x_n}$ denote a n–tuple of variables, and $\vec{t_n}$ a n–tuple of terms, the program $pl2cut(P)$ is obtained by:*

$$pl2cut(D.P) = pl2cut(cut(D), D).pl2cut(P)$$
$$pl2cut(\epsilon) = \epsilon$$

where

$$cut(C.D) = \begin{cases} 1 & \textit{if } cut(C) \\ 0 & \textit{otherwise} \end{cases}$$

and

$$pl2cut(0, C.D) = \begin{cases} \texttt{p}(\vec{x_n})\texttt{:- pN}(\vec{x_n}).pl2cut(1, rename(\texttt{pN}, C.D)) & \textit{if } cut(C) \textit{ and} \\ & \quad head(C) = \texttt{p}(\vec{t_n}) \\ C.pl2cut(0, D) & \textit{if } \neg cut(C) \end{cases}$$

$$pl2cut(1, C.D) = \begin{cases} \texttt{p}(\vec{x_n})\texttt{:- pN}(\vec{x_n}).pl2cut(0, rename(\texttt{pN}, C.D)) & \textit{if } \neg cut(C) \textit{ and} \\ & \quad head(C) = \texttt{p}(\vec{t_n}) \\ unravel(C).pl2cut(1, D) & \textit{if } cut(C) \end{cases}$$

$$pl2cut(N, \epsilon) = \epsilon$$

■

The resulting program $pl2cut(P)$ is obviously equivalent to P, as the foldings preserve its semantics (as shown for example in [TS84]), and the semantics of cut is also preserved (by preserving its scope in all cases). The remaining part to translate cut to AKL is to rewrite "`!`" into "`->`." We have already mentioned that cut exists in AKL, and that this operator does not require quietness of its guard to proceed. However, we will prefer to use the conditional for translation purposes. This is due to the experimental evidence (which will be shown in Section 7) that the conditional shows performance advantages over the "noisy" cut.

In order to guarantee correctness when using conditional instead of cut, we have to guarantee quietness of the guards when solved. Quietness of a solved guard is achieved if its solution does not add constraints to variables outside the guarded goal, other than those which already appear in its environment. Entailment of the guard solutions by their environment could be checked (by an analysis of the program) to guarantee this. Instead, we prefer a simpler solution, where all constraints possibly added on external variables by the guard are delayed until after the guard. In the examples shown, quietness is straightforward, as no constraints (bindings, in those cases) appear before the cut (assuming that `test(X)` does not further constrain `X`). If this is not the case, constraint telling has to be made explicit in the form of an equality constraint (a unification) and placed after the cut itself, i.e. outside the guarded part of the clause.

Example 3 *Consider that* `Y` *is further instantiated by the* `output/1` *predicate in the CLP program to the left. The corresponding AKL program to the right makes the corresponding conditional quiet.*

```
p(X,Y):- test(X), output(Y), !.
p(X,Y):- s(X,Y).
```

```
p(X,Y):- test(X), output(Y1) -> Y1=Y.
p(X,Y):- s(X,Y).
```

Note that knowledge of input/output modes of variables is required for performing this transformation, and that the transformation may not always be safe.[2] Safety can be guaranteed by employing existing techniques for semantic analysis of the CLP program, as we will illustrate. When safe, such transformation can indeed make quiet an otherwise noisy pruning. What it does is to delay "output" constraint telling until the guard is promoted by making it explicit in the body part of the clause.

We regard a variable to be *output* in a query if execution for this query will further constrain it; a variable will be regarded as *input* if execution will depend on its state of instantiation (or constraint). In other words, a variable is an output variable in a literal if it is further constrained by the query this literal represents, it is an input variable if it makes a difference for the execution of the literal whether the variable is constrained or not.[3] Note that a given variable can be both input and output, or none of them.

Definition 5 (input and output variables) *Given a CLP program P, and a literal g of a clause C of P, a variable $x \in vars(g)$ is an input variable if for any derivation*

[2]Note also that this transformation, when safe, may be of advantage as well in standard CLP compilers in order to avoid trailing overhead.

[3]These definitions are similar to those independently proposed in [SCWY91], (and also in the spirit of those of Gregory [Gre85]), which describes translation techniques from Prolog to Andorra-I, an implementation of the Basic Andorra Model. Although the techniques used in such a translation have some relationship with those involved in Prolog-AKL (and CLP-AKL) translation, the latter requires in practice quite different techniques due to AKL being based on the *Extended* Andorra Model (thus having to deal with the possibility of parallelism among non–determinate goals and the stability rules) and the rather different way in which the control of the execution model (explicit in AKL and implicit in Andorra-I) is done in each language.

of P, say

$$s_0 \rightarrow \ldots \rightarrow \langle (g\sigma, g_i), \theta_i \rangle \rightarrow \ldots \rightarrow \langle g_i\theta_{i+n} \rangle$$

by replacing g with $g[x/y]$ where y is a completely new variable, we obtain a derivation with the same derivation steps

$$s_0 \rightarrow \ldots \rightarrow \langle (g\sigma[x/y], g_i), \theta_i[x/y] \rangle \rightarrow \ldots \rightarrow \langle g_i\theta_{i+n}[x/y] \rangle$$

Variable x is an output variable if for any derivation of the form above it happens than $\theta_{i+n}|_x$ entails $\theta_i|_x$. ■

The objective of a transformation such as the one proposed is to rename apart all output variables in the head of a pruning clause, and then bind the new variables to the original ones in the body of the clause, leaving input variables untouched. In general, it is unwise to rename apart input variables since, from their own definition, this renaming would make the variable appear unconstrained and potentially result in growth in the search space of the goals involved. This would not meet our objective of preserving the complexity of the program (and perhaps not even that of preserving its semantics).

However, since a variable can be both input and output a conflict between renaming and not–renaming requirements appears in such cases. For the cases in which a variable cannot be "moved" after the guard operator, the translation has to default to the noisy cut operator. It is necessary that every noisy cut be sequentialised, by using the AKL sequential composition operator (`&`[4]). This is to ensure that pruning would occur in the same context that it would in CLP. Thus, every call to the pruning predicate has to be sequentialised to its right, and every call to a predicate sequentialised has in turn to be also sequentialised. For this reason noisy pruning is not very efficient, and thus the translation tries to minimise its use.

The final step of the transformation for cuts will then try to rewrite every cut into an AKL conditional, possibly renaming apart output variables, based on the knowledge of input/output modes in the CLP program. Let us now consider $input(x)$ to be true if x is known to be an input variable, $output(x)$ if an output variable.

Algorithm 4 (quiet cuts) *Given a CLP program P, cuts are made quiet by transforming P into $cut2akl(P)$ as follows. Let $x_{m,n}^{\rightarrow}$ denote a tuple of variables x_i with $i \in (m, n]$.*
$cut2akl(C.P) = cut2aklc(C).cut2akl(P)$
$cut2akl(\epsilon) = \epsilon$
where
$cut2aklc(C) = C$ *if* $\neg cut(C)$, *otherwise*

[4]This operator inhibits concurrent execution, and can be viewed as a guard. For example, it can be defined as: `A & B :- A ? B.`

$$cut2aklc(\mathtt{p}(\vec{x_n}\vec{y})\text{:- } L(\vec{x_n}\vec{z}), !, R(\vec{y}\vec{z})) = \begin{cases} \mathtt{p}(\vec{x_n}\vec{y})\text{:- } L(\vec{w_m}\vec{x_{m,n}}\vec{z})\text{->}\vec{x_m} = \vec{w_m}, R(\vec{y}\vec{z}) \\ \quad \textit{if } \forall i \in [1,m] output(x_i) \wedge \neg input(x_i) \\ \quad \textit{and } \forall i \in [m,n] \neg output(x_i) \\ \mathtt{p}(\vec{x_n}\vec{y})\text{:- } L(\vec{x_n}\vec{z})\,!\,R(\vec{y}\vec{z}) \\ \quad \textit{if } \exists i \in [1,n] output(x_i) \wedge input(x_i) \\ \mathtt{p}(\vec{x_n}\vec{y})\text{:- } L(\vec{x_n}\vec{z})\text{->}R(\vec{y}\vec{z}) \\ \quad \textit{if } \forall i \in [1,n] \neg output(x_i) \end{cases}$$

. ∎

Note that the knowledge assumed in the transformation requires in general a global analysis of the program and can only be approximated. In order for the translation to be correct, conservative approximations have to be made. Thus, some variables may be regarded as being input (resp. output) when they are not. For these (and only these) variables, an automated translation, if interactive, could "ask" the user for the kind of transformation to be performed. If a non–interactive translation is preferred, these cases default to those of the above transformation algorithm.

We will consider the issue of program analysis in Section 6. At this point, given that the analysis is conservative in the sense just described, we obtain the following result.

Theorem 2 (AKL and CLP program equivalence) *Given a CLP program P, the computations of its transformed AKL program $cut2akl(pl2cut(P))$ give the same solutions than those of P.*

Proof: (Outline) Consider an AKL computation in which choice split is always performed on the leftmost choice–box of the configuration. Since the analysis of the CLP program is conservative, any application of a conditional rule is enabled for a quiet guard. For a noisy guard, any application of the noisy cut rule is enabled in a configuration equivalent to the CLP environment, due to sequentialisation. The choice split rule follows Theorem 1. Therefore the AKL program $cut2akl(pl2cut(P))$ obtains the same solutions. ∎

4.2 Synchronisation of Side–effects

In general, the purpose of side–effect synchronisation is to prevent a side effect from being executed before other preceding (in the sense of the sequential operational semantics) side–effects or goals, in the cases when such adherence to the sequential order is desired. In our context, if side–effects are allowed within (parallel) AKL code and a behaviour of the program identical to that observable on a sequential CLP implementation is to be preserved, then some type of synchronisation code should be added to the program. In general, in order to preserve the sequential observable behaviour, side–effects can only be executed when every subgoal to their left has been executed, i.e. when they are "leftmost" in the execution tree. However, a distinction can be made between *soft* and *hard* side–effects (a side–effect is regarded to be *hard* if it could affect

subsequent execution), see [DeG87] and [MH89]. This distinction allows more parallelism. It is also convenient in this context to distinguish between side–effect builtins and side–effect procedures, i.e. those procedures that have side–effects in their clauses or call other side–effect procedures.

To achieve side–effect synchronisation, various compile–time methods are possible:

- To use a chain of variables to pass a "leftmost token", taking advantage of the suspension properties of guards to suspend execution until arrival of the token [SCWY91].
- To use chains of variables as semaphores with some compact primitives that test their value. In [Mut91] a solution was proposed along such lines, and its implementation discussed.
- To use a sequentialisation builtin to make the side–effect and the code surrounding it wait; this primitive would be in our case the sequentialisation operator "&".

In the first solution, a pair of arguments is added to the heads of relevant predicates for synchronisation. Side–effects are encapsulated in clauses with a wait guard containing an ask unification (being quiet) of the first argument with some known value (*token*), to be passed by the preceding side–effect upon its completion. Upon successful execution of the current side–effect the second argument is bound ("told") to the known value and the token thus passed along. This quite elegant solution can be optimised in several cases.

The second solution can be viewed as an efficient implementation of the first one, which allows further optimisation [Mut91]. The logical variables which are passed to procedures in the extra arguments behave as semaphores, and synchronisation primitives operate on the semaphore values.

In the third solution, every soft side–effect is synchronised to its left with the sequentialisation operator, and every hard one both to its left and right. This sequentialisation is propagated upwards to the level needed to preserve correctness. This introduces some unnecessary restrictions to the parallelism available. However, if side–effects appear close to the top of the execution tree, this may be quite a good solution.

5 Achievement of Independent And-Parallelism

In order to achieve more parallelism than that available by the translations described so far one might think of translating CLP into AKL so that every subgoal could run in parallel unrestricted. However, this can be very inefficient and would violate the premise of preserving the results and complexity of the computation expected by the user. On the other hand, and as mentioned before, parallel execution of *independent* goals, even if they are nondeterminate, is an efficient and desirable form of parallelism and its addition

motivated the development of the EAM, on which the AKL is based. Nevertheless, in AKL goals known to be independent have to be explicitly rewritten in order to make sure that they will be run in parallel. This is because of the rules that govern the (nondeterminate) promotion, that is, the stability condition on nondeterminate promotion, which will prevent these goals for being promoted if they try to bind external variables for output. Therefore, one important issue is the transformation that is needed to avoid suspension of independent goals. Also, independence detection can and will be used to reduce stability checking, a potentially expensive operation.

Clearly, an important issue in this context is how stability/goal independence is detected. We have already presented the necessary machinery for this in previous chapters. At this point we can regard our translation as a two step process. For any CLP clause C the first step yields $annotate(C)$ (see [dlBBH95]), which is then transformed as presented below in a second step, while clauses in the same definition are transformed as presented in previous sections. The result of $annotate(C)$ is given as a CIAO[HtCg94, Bue95, CH95] clause, i.e. the program itself expresses which goals are independent and under which conditions. These conditions are expressed in the form of if-then-elses (which will then have to be folded out) and parallelism itself is made explicit by using the "`&`" operator to denote parallel conjunction instead of the standard sequential conjunction denoted by ",".[5] For our purposes we can consider CIAO restricted to its subset which supports CLP plus the parallel conjunction operator "`&`" and the builtins `def/1` and `unlinked/2` which are used in conditional parallel expressions (see [dlBBH95]). Some new issues are involved in the interaction between the conditions of these parallel expressions and other goals run in parallel concurrently, as it would be the case in AKL.

At this point the CIAO conditionals are regarded as input to the translator. As such, if-then-elses are pre–processed in the form mentioned in the previous sections and the remaining issue is the treatment of the parallelisation operator "`&`". In implementing this operator we will use the AKL property that allows local and unrestricted execution of guards, i.e. goals that are encapsulated in a guard can run in parallel with goals in other guards even if they are nondeterminate. The transformation that takes advantage of this will (1) put goals known to be independent in (different) guards, and (2) extract output arguments from the guards, binding them in the body part of the clauses; the last step being required so that the execution of these goals is not suspended because of their attempt to perform output unification, as stability requires. With the guard encapsulation we ensure that those predicates will be executed simultaneously and independently. The following example illustrates the transformation involved.

Example 4 *Encapsulation of independent subgoals*

```
p(X):- (def(X),                          p(X):- pp(X,Y,Z), s(Y,Z).
        unlinked(Y,Z) ->
```

[5] Note that in AKL these operators have just the opposite meaning!

```
   q(X,Y) & r(X,Z)
 ; q(X,Y) , r(X,Z)
),
s(Y,Z).
```

```
pp(X,Y,Z):- def(X), unlinked(Y,Z) ->
            qp(X,Y), rp(X,Z).
pp(X,Y,Z):- q(X,Y), r(X,Z).

qp(X,Y):- q(X,Y1) ? Y=Y1.

rp(X,Z):- r(X,Z1) ? Z=Z1.
```

When the condition is met, both subgoals will be tried by the reduction rule, then *both guards* will be completely and locally solved, and then, as goals are independent (because the condition met) and no output is produced in the guard (because of the back–bindings introduced), the choice splitting rule is always applicable and all solutions of guards will be tried in the standard Cartesian product way. Thus, parallel execution is ensured for those goals that are identified as independent.

On the other hand, when the condition fails (the goals being dependent) they appear together in a body with an empty guard. This means that the guard will be immediately solved, the clause body promoted, and subgoals tried simultaneously. Then the standard stability and promotion rules will apply.

Algorithm 5 (encapsulated goals) *Given a CIAO program P, its parallel expressions are encapsulated in AKL guards by transforming it into a program $pl2pp(P)$. The transformation is defined by the mapping $pl2pp$: CIAO $\rightarrow$ AKL as follows:*
$pl2pp(C.P) = ppl2ppc(C).pl2pp(P)$
$pl2pp(\epsilon) = \epsilon$
where
$pl2ppc(\mathtt{p}) = \mathtt{p}$
$pl2ppc(\mathtt{p}\text{:-}\ B) = \mathtt{p}\text{:-}\ B'.C'$ *if* $\langle B', C' \rangle = pl2ppb(B)$
and
$pl2ppb(A, B) = \langle A', B', C_A.C_B \rangle$ *if* $\langle A', C_A \rangle = pl2ppb(A)$ *and* $\langle B', C_B \rangle = pl2ppb(B)$
$pl2ppb(A\&B) = \langle A', B', C_A.C_B \rangle$ *if* $\langle A', C_A \rangle = encap(A)$ *and* $\langle B', C_B \rangle = pl2ppb'(B)$
$pl2ppb(\mathtt{g}) = \mathtt{g}$
and
$pl2ppb'(A, B) = encap(A, B)$
$pl2ppb'(A\&B) = pl2ppb(A\&B)$
$pl2ppb'(\mathtt{g}) = encap(\mathtt{g})$
and
$encap(A, B) = \langle \mathtt{p_N}, ppg(\mathtt{p_N}\text{:-}\ A').C \rangle$ *if* $\langle A', C \rangle = pl2ppb(A, B)$
$encap(\mathtt{g}) = \langle \mathtt{p_N}, ppg(\mathtt{p_N}\text{:-}\ \mathtt{g}) \rangle$
and
$ppg(\mathtt{p}\text{:-}\ B) = \mathtt{p}\text{:-}\ L?R$ *if* $cut2aklc(\mathtt{p}\text{:-}\ B, !, true) = \mathtt{p}\text{:-}\ L\%R$ *for some guard* $\%$ ∎

It should be noted that, as in the case of cut, and in addition to detecting goal independence, to be able to perform this transformation it is necessary to have inferred mode information regarding the predicate clauses.

5.1 Cohabitation of Dependent and Independent And-Parallelism and Stability Checks

When evaluating the conditions of parallel expressions at run–time within a parallel framework such as that of the AKL, they may not evaluate to the same value than during a fork/join execution such as that of CIAO (for the case of the goal–level parallel expressions we consider). This is what has been termed in another context the *CGE-condition problem* [GSCYH91][6], and may result in a loss (or increase) of parallelism. To deal with these issues, different levels of restrictions can be placed on the translation:

- Disallow any parallel execution except for those goals found to be independent.
- Allow parallel execution only for goals not binding variables that appear in the conditions or CGE (parallel expressions).
- Allow parallel execution outside a CGE but sequentialise before and after the conditional parallel expressions.
- Allow unrestricted parallel execution unrestricted, i.e. no sequentialisation is to be done.

The first solution can be implemented by translating every conjunction as a *sequential* AKL conjunction, except those joining independent goals. This will lead to a type of execution where only goals known to be independent are run in parallel and which directly resembles that of CIAO fork/join [HG90]. The same search space as CIAO will be explored. Nondeterminate (and determinate) promotion will then be restricted to only independent and sequential goals. Thus, one very important advantage of this translation is that *no checks on stability ever need to be done*, as stability is ensured for sequential and independent execution. This is an important issue since stability checking is a potentially expensive operation (and very closely related to independence checking). Thus, in an *ideal* AKL implementation code translated as above, i.e. free of stability checks, should run with comparable efficiency to that of CIAO. On the other hand, the transformation loses determinate dependent and–parallelism and its desirable effect of co–routining, which could be useful in reducing search space [SCWY90].

The second solution attempts to preserve the environment in which the CGE evaluates while allowing co–routining of goals that don't affect CGE conditions and goals. Although interesting, this appears quite difficult to implement in practice as it requires very sophisticated compile–time analysis and will probably incur in run–time overheads for checking of the conditions placed in the program.

The third solution can be viewed as a relaxation of the first one to achieve some co–routining, or as an efficient (and feasible) way of partially implementing the second one.

[6] Note that some other problems mentioned in [GSCYH91] regarding the interaction between independent and dependent and–parallelism (in particular, the *determinate goal problem*) are less of an issue in the proposed translation to AKL because independent goals execute in their own environments, thanks to the dynamic scoping of AKL guards. In any case, the AKL implementation is assumed to cope with all types of goal activations possible within the EAM.

Goals before and after are allowed to execute in parallel using the Andorra Principle, but they are sequentialised just before and after a CGE. In this way CGEs evaluate in the same context as in the execution of CIAO and the same level of independent and–parallelism is achieved. This translation has the good characteristics regarding search space of the previous one. In addition, some reduction of search space due to co–routining will be achieved. However, stability checking, although reduced, cannot in general be eliminated altogether.

The fourth solution will allow every goal to run in parallel. The full EAM and AKL operational semantics (including stability) has to be preserved. Independence checks may fail where they wouldn't in CIAO (therefore losing this parallelism), but also succeed where they would fail in CIAO (therefore gaining this parallelism). Also, the number of parallel steps will always be equal to or less than in CLP (although different than in CIAO). This solution, the first, and the second ones appear as quite reasonable compromises and offer different tradeoffs. The current translation approach uses this fourth option.

6 The Role of Program Analysis

We have mentioned the need for inferring modes of clause variables (i.e. whether they are input or output variables) in CLP programs for the type of translation we pursue. The main reason for this need is that output variables in a clause have to be identified in order to rename them apart and place corresponding bindings for them in the body part of the clause. This is needed in the case of pruning clauses, and also in the transformed clauses for parallel execution, and it is captured in the definition of the mapping *cut2aklc* in Algorithm 4. We will now discuss the usefulness of the program analysis technology for CLP [dlBH93, DJBC93, dlBHB$^+$94], and in particular, that discussed in [dlBBH95], in our translation.

Recall that a program variable (or an argument) is output in a literal if the call to the corresponding predicate further constrains this variable, and it is input in a literal if its constraint state is going to be checked in the execution of the call for that literal. Therefore, in the translation process information on the *state* of a variable in the constraint store is essential for determining input/output arguments. This we can show by simply expressing the input/output character of variables in terms of a lattice of its possible states. Table 1 shows how the input or output character of variables can be decided in a good number of cases based on the information directly available for a literal g_i from a global analysis over such a lattice. Recall that such global analyses are based on a collecting semantics with abstract substitutions λ_i at program points i corresponding to a state *prior* to the execution of g_i. We denote "Def" the character of a variable which is definitely constrained to a unique value. "Unc" denotes a variable not constrained by the store, and "Con" a variable whose domain is constrained.

Arguments of a literal inherit the input/output character of its variables. The rel-

λ_i	λ_{i+1}	Output?	Input?
def	(def)	no	*
unc	unc	no	*
	con	yes	no
	def	yes	no
con_1	con_1	no	*
	con_2	yes	?
	def	yes	?

Table 1: Input/output variables

evance of an analysis yielding information on unconstrained variables (such as that based on the Fr abstract domain of [DJBC93, dlBBH95]) in detecting input/output variables is clear. From the table we identify cases in which the variable is known not to be an input variable, without any further analysis (i.e. when the variable is unconstrained). Furthermore, we realize that if a variable is known not to be an output variable then it doesn't need to be renamed apart and it is not necessary to determine whether it is an input variable or not ("*" cases). Knowing that a variable is definitely constrained to a unique value also helps in this. Thus, the benefits of an analysis yielding both classes of information is clear. This is the case of the FD abstract domain of [dlBHB$^+$94, dlBBH95]. Reducing the situations where knowing if a variable is input is quite useful since inferring whether a variable binding is needed or not requires additional analysis ("?" cases). This analysis seeks to decide if a variable is crucial in clause selection or checking. Note that the analysis has to be extended for every child procedure of the one being analysed.

7 Experimental Results

This section presents some results on the timing of a number of benchmarks using our translation in the prototype AKL system. The AKL versions of the programs obtained through automatic compile–time translation are compared with versions specifically written for AKL. Timings for the sequential versions of the programs are also included for comparison and also with the intention of identifying translation paradigms that help efficiency. With this aim in mind, the set of benchmarks has been chosen so that performance results are obtained for several different programming paradigms, and a number of different translation issues are taken into account. The results show that translation suffices in most cases, provided state–of–art analysis technology is used.

Since our translation is parametric on the underlying constraint system of the program, and our experiments were not to be influenced by this, availability restrictions forced us to restrict the experiments to the Herbrand constraint system. Timings have been obtained for the sequential program (compiled to native code), the AKL program resulting from automatic translation and the "hand-written"-AKL version. SICStus 2.1 #9 and a sequential AKL prototype system, AGENTS 1.0, from SICS, have been used.

Times have been obtained on a SunOS SPARC station, and correspond to execution of the program until the first solution is found. They are an average of ten consecutive executions done after a first one (not timed) and are given in milliseconds, rounded up to tens.

Benchmark Program	SICStus	AKL	
		translated	h. written
qsort (1st)	10	10,500	80
qsort	10	70	80
mergesort	10	450	500
money (1st)	14,950	50,390	540
money	11,830	50,340	540
zebra	2,330	10,070	1,800
scanner	262,550	350	80
triangle	990	58,550	6,820

Table 2: Timings for direct translations.

We briefly introduce the programming paradigms represented by each of the benchmarks used. Qsort has been translated in two ways, one that "folds" pruning definitions, and another one that is able to "extend" the cut to all clauses, the latter showing an advantage w.r.t. the former. Mergesort illustrates the advantage of being able to detect that some cuts are not noisy (as opposed to defaulting to noisy cut in every case). In fact, in this case the translated version is slightly faster than the hand–coded one!

For money we have used two different versions. In the first version of the program the problem is solved through extensive backtracking. In the second one the ordering of goals is improved in the sequential program. As in zebra the difference with the "hand-written" version is in the use of the arithmetic predicates: addition is programmed in the hand–coded AKL version as illustrated by the following `sum/3` predicate, in which the co–routining effect provides a "constraint solving" behaviour:

```
sum(X,Y,Z):- Z0 is X+Y | Z = Z0.
sum(X,Y,Z):- X0 is Z-Y | X = X0.
sum(X,Y,Z):- X0 is Z-X | Y = Y0.
```

Scanner is a program where AKL can take a large advantage from concurrent execution and the "determinate–first" principle, even without explicit control, and this is shown in the good performance of the translated program. The sequential programming of this benchmark is clear and straightforward, but is not very efficient executed sequentially. On the other hand, in triangle, heavy use of special AKL features has been made, through hand–optimisation.

In matrix, hanoi, query, and maps (and also qsort), encapsulation of different classes of goals has been tried. The results show that encapsulating independent goals which are determinate provides no improvement, but performance improves when they are

nondeterminate. Performance also improves in the case of goals which act in a producer/consumer fashion (maps). These results suggest that AKL control similar to that of hand–coded versions can be imposed automatically for situations other than independence of goals (such as non–determinate goals or producers of values for variables).

Benchmark Program	SICStus	AKL translated	
		with encap.	direct
qsort	10	90	80
matrix	30	190	120
hanoi	10	20	350
query	10	160	230
maps	30	90	1,780

Table 3: Timings for encapsulation of goals.

The automatic transformation achieves reasonably good results when compared to code specifically written for AKL, provided one takes into account that the starting point is a sequential program with little specification of control, and it is being compared to an AKL program where control has been optimised by the programmer. The examples where the largest differences show are those in which the control imposed by hand in the AKL program changes the complexity of the algorithm, generally through smart use of suspension (as in the `sum/3` predicate), something that the current transformation can not do automatically. However, the results also show that it would obviously be desirable to extend the translation algorithms towards implementing some of the smart forms of control that can be introduced by a good AKL programmer.

When comparing with SICStus, the figures show advantage in running the programs in AKL. The results show that a variable performance improvement can be obtained whenever determinism is significant in the problem (this is quite spectacular in scanner). When automatically translating the sequential algorithms, running in AKL incurs in a certain overhead. However, the encapsulation transformation can help efficiency in some cases.

8 Conclusions

The translation scheme presented benefits from the power of a language such as AKL, which embeds both the concurrent and search–based programming paradigms, as well as from transformation techniques based on independence. On the one hand, it allows bridging the differences between CLP and AKL, and also achieving full exploitation of the (and-)parallelism possibilities of the target language, while offering the programmer the more familiar semantics of sequential logic programming. On the other hand, previous results on goal independence guarantee that program complexity can be preserved in the parallel execution of the transformed program.

The transformation is relevant even in the case of a sequential AKL implementation since the reduction of stability checking which follows from knowledge of goal independence can already be of significant advantage (if it can be transmitted to the compiler), given the expected cost of stability tests. In the case of a parallel AKL implementation the transformation amounts to a form of automatic parallelisation and search space reducing implementation for CLP programs which exploits the EAM, and imposes a particular form of control on it. The advantages of considering independence notions to deal with stability in the AKL computation have been further elaborated in [MD94]. Our transformation can be viewed as a means of (partially) dealing with this subject at compile–time.

The efficiency results obtained for a sequential implementation of AKL (over the Herbrand domain) do not allow us to state the expected efficiency of the translation in other situations. Nevertheless, they point out a positive trend in the exploitation of independence which could become effective in a parallel implementation of AKL.

In any case, and regarding automation of the translation, an effective application of a translator can be found in a programming environment where the user writes down CLP code for problem solving, taking advantage of its declarativeness and its clear semantics, and uses AKL where he/she wants to impose particular control or make concurrency explicit. The translator will then transparently allow the whole code to run on AKL. This is realized in part in the CIAO compiler, which, using techniques such as those described herein, can support different programming modes simultaneously via program transformation.

References

[BH92] F. Bueno and M. Hermenegildo. An Automatic Translation Scheme from Prolog to the Andorra Kernel Language. In *Proc. of the 1992 International Conference on Fifth Generation Computer Systems*, pages 759–769. Institute for New Generation Computer Technology (ICOT), June 1992.

[Bue95] F. Bueno. The CIAO Multiparadigm Compiler: A User's Manual. Technical Report CLIP8/95.0, ACCLAIM Deliverable D3.2/3-A4, Facultad de Informática, UPM, June 1995.

[CH95] D. Cabeza and M. Hermenegildo. Distributed Concurrent Constraint Execution in the CIAO System. Technical Report CLIP14/95.0, ACCLAIM Deliverable D4.3/2-A2, Facultad de Informática, UPM, June 1995.

[Col90] A. Colmerauer. An Introduction to Prolog III. *CACM*, 28(4):412–418, 1990.

[DeG87] D. DeGroot. Restricted AND-Parallelism and Side-Effects. In *International Symposium on Logic Programming*, pages 80–89. San Francisco, IEEE Computer Society, August 1987.

[DJBC93] V. Dumortier, G. Janssens, M. Bruynooghe, and M. Codish. Freeness Analysis in the Presence of Numerical Constraints. In *Tenth International Conference on Logic Programming*, pages 100–115. MIT Press, June 1993.

[dlBBH95] M. García de la Banda, F. Bueno, and M. Hermenegildo. Automatic Compile-Time Parallelization of CLP Programs by Analysis and Transformation to a Concurrent Constraint Language. Technical Report CLIP3/95.0, ACCLAIM Deliverable D3.3/3-A1, Facultad de Informática, UPM, June 1995.

[dlBG94] María José García de la Banda García. *Independence, Global Analysis, and Parallelism in Dynamically Scheduled Constraint Logic Programming.* PhD thesis, Universidad Politécnica de Madrid (UPM), July 1994.

[dlBH93] M. García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 437–455. MIT Press, Cambridge, MA, October 1993.

[dlBHB+94] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. Draft, 1994.

[dlBHM94] M. García de la Banda, M. Hermenegildo, and K. Marriott. Search Space Preservation in CLP Languages. Technical Report CLIP11/94.0, University of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, September 1994. Also provided as attachment of deliverable D3.2-3.3/2.

[Gre85] S. Gregory. *Design, Application and Implementation of a Parallel Logic Programming Language.* PhD thesis, Imperial College of Science and Technology, London, England, 1985.

[GSCYH91] G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: A Model Intergrating Dependent-, Independent-, and Or-parallelism. Technical report, University of Bristol, March 1991.

[Hen89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[HG90] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[HJ90] S. Haridi and S. Janson. Kernel Andorra Prolog and its Computation Model. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 31–46. MIT Press, June 1990.

[HR95] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

[HtCg94] M. Hermenegildo and the CLIP group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, LNCS 874, pages 123–133. Springer-Verlag, May 1994.

[Jan94] Sverker Janson. *AKL. A Multiparadigm Programming Language.* PhD thesis, Uppsala University, 1994.

[JL87] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *ACM Symp. Principles of Programming Languages*, pages 111–119. ACM, 1987.

[MD94] R. Moolenar and B. Demoen. Full parallel search in AKL. ACCLAIM Deliverable D4.1/2-3, Dept. Computer Science, K.U. of Leuven, September 1994.

[MH89] K. Muthukumar and M. Hermenegildo. Complete and Efficient Methods for Supporting Side Effects in Independent/Restricted And-parallelism. In *1989 International Conference on Logic Programming*, pages 80–101. MIT Press, June 1989.

[Mut91] Kalyan Muthukumar. *Compile-time Algorithms for Efficient Parallel Implementation of Logic Programs.* PhD thesis, University of Texas at Austin, August 1991.

[SCWY90] V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, April 1990.

[SCWY91] V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.

[TS84] H. Tamaki and M. Sato. Unfold/Fold Transformations of Logic Programs. In *Second International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, 1984.

[War90] D.H.D. Warren. The Extended Andorra Model with Implicit Control. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, Box 1263, S-163 13 Spanga, SWEDEN, June 1990. SICS.